THE IMPLEMENTATION OF A SUBSET OF PROCEDURES IN AN ALGOL 68 COMPILER

By

ALAN DANIEL EYLER

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1970

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1975

THE IMPLEMENTATION OF A SUBSET OF PROCEDURES IN AN ALGOL 68 COMPILER

Thesis Approved:

J. E. Hedrick
Thesis Adviser

D D Fisher

K W Grace

N N Durham
Dean of the Graduate College

PREFACE

This project is concerned with the implementation of a subset of the procedure facility in the existing compiler for the programming language ALGOL 68 at Oklahoma State University. The primary objective is the enhancement of the compiler to support procedure constants and their invocation through calls as prescribed by the syntax and semantics of the language definition. Secondary objectives include the solution of space problems imposed by the implementation of the compiler on a mini-computer.

I wish to express my deep appreciation to my advisor, Dr. G. E. Hedrick, for his continuous guidance and encouragement. I also wish to thank the entire faculty of the Computing and Information Sciences Department who made my time at OSU extremely enjoyable as well as informative. A special thanks is extended to Dr. D. D. Fisher and Dr. D. W. Grace who not only made my study at OSU possible, but gave the much needed encouragement when the going got tough. Lastly, thanks to Pam Haught who spent those many hours retyping the passages which I thought would "sound better this way."

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Objective

ALGOL 68 is a very powerful, multipurpose high level computer language designed to supply the programmer with the most desirable features of ALGOL 60, PL/1, FORTRAN and COBOL as well as certain features not available in any other programming language. In 1973 John C. Jensen implemented a translator for a scientific subset of ALGOL 68 to be used an as instructional tool at Oklahoma State University (7). This translator supported basic operations on the primitive modes (10) int, real, and bool as well as compl; some limited character manipulations (mode char); and formatless input and output. Since that time several features have been added to the translator by students at OSU. However, one very desirable feature of the language, the ability to support the construct termed procedures, was still not available. Therefore, it was decided to enhance the present translator to support a portion of this ALGOL 68 feature. Several restrictions of the feature were included in the implementation design. These are described in Chapter II. However, the design included a sufficient number of the facilities of the language

definition to provide the programmer with a useful aid to assist in the accomplishment of his task.

History of ALGOL 68

During the 50's many initial efforts were made to develop what are now called high level language. Perhaps the most enduring of these has been FORTRAN. The first ALGOL language, ALGOL 58, was developed through the joint efforts of the Association for Computing Machinery and the Association for Applied Mathematics and Mechanics and formalized in June, 1958 (21).

Since these early beginnings there has been a steady trend toward more use of high level languages. As man's understanding of language syntax and semantics has increased, so has his ability to define useful programming languages. Jean E. Sammet gives a very detailed history of the development of programming languages in the United States (21).

ALGOL 68 represents a continuous development process dating back to the realization of the desirability of a common programming language for communication of ideas between people as well as between people and computers. ALGOL 58 was the original result of this effort. With attempts at implementation, many omissions and ambiguities were discovered in the language. This, aided by the development of a formal method of defining syntax now known as the Backus Normal Form (1), resulted in the

definition of a revised language known as ALGOL 60 (13). Although this was a much improved language, errors arose as they always do and the "Revised Report on the Algorithmic Language ALGOL 60" (14) was issued following a review meeting of the authors of the language in 1962 at Rome. At this time responsibility for the language was transferred to the Working Group on ALGOL (WG2.1) of the International Federation for Information Processing. This group has had continuing responsibility for the development of ALGOL since that time.

In exercising this responsibility, WG2.1 has considered extensions and improvements of the language since 1963. In 1965 three language descriptions were proposed. Two of these used the Backus Normal Form for description of the proposed grammar. The third by Aad van Wijngaarden introduced a new descriptive method which has become known as a van Wijngaarden Grammar. Following this meeting Dr. van Wijngaarden, working with B. J. Mailloux, J. E. L. Peck, and C. H. Koster, revised and improved the definition to the form finally accepted by WG2.1 in December, 1968 (23).

Since 1968 several supporting documents have been published (16,10). Much attention has been given to evaluating and implementing the language as defined in what is known among ALGOL groups as "The Report". Almost annual conferences on implementation (17,18,19,20) have been held to exchange information concerning difficulties

encountered and methods developed in implementing ALGOL 68 compilers.

WG2.1 decided to publish a revised final definition of the language ALGOL 68 (24) after consideration of the many recommendations coming out of these conferences and those made by other people interested in the language. The document was approved at a meeting in Los Angeles in 1973 (9). This is considered to be a static definition and no further changes are anticipated.

"Guide lines for a language designer
(i) provide an undecipherable description
(ii) implement a subset
(iii) include mega disaster possibilities
(iv) inform users of their rescue from potential mini disasters
(v) imbed translator in an imnipotent system."

D. Fisher

CHAPTER II

PROBLEM SPECIFICATION

Preliminary

One of the persistent motivations in the development of all the ALGOL language has been to produce a common, machine-independent programming language for the user. To this end the definition of ALGOL 68 (23) incorporates a large number of new technical terms. Some of them encompass old concepts under new words and some specify new concepts. In both cases an understanding of the terms is essential to an understanding of any discussion of the language. A list of definitions for those terms which are applicable to the materials covered in this paper are listed in Appendix A. Readers should refer to this Appendix before proceeding. This will greatly ease the task for the "uninitiated" and will be a (perhaps useful) refresher for the "inititated". Any reader who is familiar with PL/1 but not ALGOL 68 might find the comparison by Hedrick and Alexander (5) helpful in associating the new terms with the concepts to which they apply.

Review of Previous Work

The literature contains very few articles concerning the topics pertinent to this study. Only one article directly concerned with the implementation of procedures was found (12). A large portion of the literature is concerned with corrections, omissions, suggested changes, or evaluations, of the definition of the language. However, several articles dealing with proposed or existing implementations of the language or subsets of the language have been published. Ledin gives a partial list of implementation efforts as of early 1973 (8).

One of the most notable implementations is that of ALGOL 68-R at the Royal Radar Establishment in Great Malvern, England (2). This was the first working ALGOL 68 compiler. It went into limited operation in April, 1970 on the ICL 1907F computer at RRE and was intended to be the main programming language at that installation. It is a subset of the full language with some restritions arising from its one-pass construction, some from hardware limitations, and some from considered decisions that certain features were not worth their cost. Notably for this study, automatic proceduring is not allowed in ALGOL 68-R. This feature has been eliminated entirely in the Revised Report (24). No other information directly applicable to this study is given.

The ALGOL 68 compiler at the University of British Columbia is described by Manis (11). It implements all

the features of the ALGOL 68 language except those involved in parallel processing. Parallel processing is not supported by the operating system, MTS, on the IBM System/360 for which this compiler is designed. ALGOL 68-V, as it is termed by Manis, generates code for an "Execution Machine" (EM) which performs interpretive execution. This approach is similar to that taken by Jensen in designing the Oklahoma State University translator (7). The runtime organization of EM is the classical stack model as described by Gries (4) and Van Doren (22) with ad hoc measures to support features such as the heap which are not applicable to that model.

Goos (3) described some of the difficulties encountered in the design of an ALGOL 68 compiler for Rechenzentrum der Technischen Hochschule in Munich. This compiler is for a Telefunken TR4 computer. However, the problems encountered by Goos are applicable to a full set implementation and may not appear in a subset implementation due to imposed restrictions. His observation of difficulty with scopes of references is appropriate to the present study but is negated by the mode restrictions already existing in the OSU implementation.

Jensen's paper (7) contains the original description of the implementation with which the enhancements of this effort are concerned. His paper includes a suggested method for parsing procedure declarations. The method has been used where compatible with the design of the features

to be added. For instance, routine denotations are parsed by the subprogram for analyzing unitary clauses. Jensen's grammar for procedures requires the explicit specification of a yielding moid of void. This requirement has been incorporated in the language definition of the Revised Report (24). However, the suggested additions to the pseudo-machine instruction set have been found inadequate. Additional ones were found desirable. All added instructions are described in Chapter III.

Problem Definition

Basic Considerations

From the conception of this effort one of highest priority considerations has been to maintain the original goals established by Jensen (7). He sought to produce a small, portable, semi-machine--independent translator for a scientific subset of ALGOL 68 which would be useful in an educational environment for supplying programming experience to students receiving their initial introduction to the language. For this reason, the implementation language chosen was basic FORTRAN and the machine used was the IBM 1130 at Oklahoma State University which has 8K of core and runs under Disk Monitor System Version 2, Modification 8 (DMSV2M08), using a single 1131 Disk Storage Unit. This consideration and the restrictions it imposes have been the determining factors in many of the decisions made during design and implementation of procedures.

Additionally, it is very important that the facility provided be one which is understandable and functional from the users' point of view. The problem was limited to one which could reasonably be solved in the time allowed for this effort.

Problem Design

The objective of this project is to add the capability of handling a limited version of the language feature referred to in the Revised Report (24) by the metanotion PROCEDURE to the existing ALGOL 68 translator at Oklahoma State University. This is to include syntactic recognition and generation of the semantically equivalent code for the constructs supported as well as elaboration of the generated code.

The features implemented consist of the identity declaration of simple routine constants and the calls which cause elaboration of these routine constants. Neither procedure variables nor procedure multiples ([] proc) are supported. Deproceduring, a coercion which allows a routine-denotation to appear in-line in the code and be elaborated to yield a value of some mode or void, was not included in the effort. This supplies the user with a feature very similar to procedures in PL/1 or a combination of FUNCTION and SUBROUTINE subprograms in FORTRAN (with the added effect that scoping gives to internal procedures).

Figure 1 gives a grammar which specifies the syntax of a routine identity declaration as implemented. Appendix B contains a description of how the grammar should be read and Appendix C contains a list of the terminal symbols used in the grammar. No rules are given for identifier or unitary clause as the definition given for identifier clause is a recursive definition of the entire subset.

One notable deviation from the Revised Report on the routine text is that the defining occurence for any identifier appearing in an applied occurence in the routine text must have already appeared in the program sequence; that is, the user is not allowed to use the value possessed by an identifier until it is declared. This restriction holds even though the usage is in a routine text for which the elaboration is deferred until the appearance of a call. This restriction is necessary because declarations are not recognized in an earlier pass through the source code. It has interesting implications in the invocation of a procedure from within a routine text when the range of the procedure is not local or within the range of the routine text. Consider the program segment in Figure 2. At the time the routine for p is parsed, the declaration for q is not known. Because of the appearance of what is probably an actual parameters pack following the identifier, it could easily be assumed that q is a procedure.

routine identity declaration: proc declarer, is defined as symbol, routine text, go on symbol.

proc declarer: actual procedure declarer, identifier; procedure symbol, identifier.

actual procedure declarer: formal procedure declarer.

routine text: formal parameters pack, yielding moid, routine symbol, unitary clause.

formal parameters pack: open symbol, parameters list, close symbol; empty.

parameters list: formal parameter, identifier, parameters list; formal parameter, identifier.

formal parameter: ref symbol, mode class; mode class; formal procedure declarer.

mode class: row indicator, simple mode; simple mode.

row indicator: sub symbol, number of bounds list, bus symbol.

number of bounds list: comma symbol, number of bounds list; empty.

formal procedure declarer: procedure symbol, virtual parameters pack, yielding moid; procedure symbol, yielding moid.

virtual parameters pack: open symbol, virtual parameters list, close symbol.

virtual parameters list: virtual parameter declarer, virtual parameters list; virtual parameter declarer.

virtual parameter declarer: formal parameter.

yielding moid: row indicator, simple mode; simple mode; void symbol.

simple mode: integral symbol; real symbol; complex symbol; boolean symbol; characters symbol.

Figure 1. Grammar for Routine Identity Declarations

```
begin

      int j; real a,b,c;
      proc p=(int k) real:  q(a,b)**k;
      proc q=(real r, real s) real:  r+s;
      read((a,b));
      read (j);
      c:=p(j);
      print (c)

end
```

Figure 2. An Example of an Invalid Use of a Procedure (q) Before Its Declaration

However, that approach would eliminate the ability to detect the error which results from the omission of an operator from the source code. This difficulty is easily aleviated by reversing the order of declaration or adding one more parameter to the routine-denotation for p as illustrated in Figure 3. An equally disastrous effect results when any other mode of variable is declared after its use. Consider the user's program in Figure 4. Obviously the user wants the sum of two values (strings and catenation of strings are not yet supported), but it is not clear what modes are to be added--integral, real, or complex. Nor is it clear whether coersion is required. The problem is even worse if this unitary clause is contained within another which contains declarations of the identifiers a and b with different modes. Then the code generated at compile time will not correspond to the actual values possessed at elaboration. However, the situation is com-

```
begin

      int j; real a,b,c;
      proc p=(int k, proc (real,real) real r)
            real: r(a,b)**k;

      proc q=(real r, real s) real: r+s;
      read ((a,b));
      read (j);
      c:=p(j,q);
      print (c)

end
```

Figure 3. A Valid Alternative to Figure 2.

```
begin

      proc q=real:  a+b;
      real a,b;
      read((a,b));
      print(q)

end
```

Figure 4. An Invalid Use of Values Before They Are Declared

pletely undetectable unless a previous pass has recognized all declarations.

The implemented feature agrees with the definition of procedures in the Revised Report in requiring the appearance of the yielding moid, the mode of the returned value or void, in all cases. However, the yielding moid is restricted to those modes implemented in the unitary clause parser (ALG04). This means that names (values of mode ref amode) may not be returned from a routine. Only int, real, compl, bool, char and row-of these as well as void may be returned.

When specifying a row-of amode for the yielding moid or a formal parameter, the user is not allowed to specify any bounds information. The routine will accept what is passed and work accordingly. This should cause no programming difficulties as the standard prelude procedures to determine current bounds information are supported. The user is allowed to specify ref amode in the declaration of formal parameters in the formal parameters pack although this feature is not supported in other portions of the translator. This was necessary to insure that the elaboration of a routine resulted in effects and side effects as prescribed by the Report (24). The Report prescribes that no side effects reach the actual value location for a parameter of a non-reference mode while the effect must reach the actual value location for a parameter of a ref amode. This is effectively call-by-value and call-

by-reference, respectively, and has been accomplished in the implementation by generating a local copy of all non-reference parameters. In fact, this is the action prescribed by the Report. However, since identifiers possessing names which refer to values are not distinguished by the translator program which parses unitary clauses from identifiers which possess values, the parameter of mode amode in the routine text is treated no differently from one of mode ref amode. The user will find himself free to alter the value possessed by a parameter of mode int which he should not be able to do. However, he is assured that the value referred to by the actual parameter appearing in the call is not changed at the same time.

Figure 5 contains a grammar for the call necessary to cause elaboration of a procedure constant previously declared. Special attention is warranted by the symbol "abridged unitary clause". This symbol denotes any object

procedure call: indentifier, actual parameters pack; identifier.

actual parameters pack: open symbol, actual parameters list, close symbol.

actual parameters list: actual parameter; actual parameter, actual parameters list.

actual parameter: proc identifier; abridged unitary clause.

Figure 5. Grammar for Procedure Calls

normally classed as a unitary clause except a procedure call. The appearance of an unguarded procedure identifier in an actual parameters pack causes the routine possessed by the identifier to be passed to the invoked routine as a parameter. However, the user may easily supply the yield of a procedure invocation as an actual parameter by guarding the invocation in a closed clause as illustrated in the example in Figure 6. Naturally, objects which possess a value of mode void may not appear as actual parameters as void is not considered a valid mode for formal parameters (see Figure 1).

```
begin

        int j,k; real r;
        proc p=(int m, int n)int: n+m;
        proc q=(int m)real: sqrt(m);
        read ((j,k));
        r= q((p(j,k)));
        print (r)

end
```

Figure 6. A "Guarded" Procedure Invocation in an Actual Parameters Pack

CHAPTER III

IMPLEMENTATION

Special Considerations

In the original design of the translator, a conflict developed between the desire to keep the compiler machine-independent and the desire to keep it small. A compromise had to be reached. This consisted of using a fairly universal and machine-independent language--basic FORTRAN--for the implementation language, but allowing the use of a completely machine-dependent facility, a call to the system subroutine LINK between phases. Additionally, size considerations forced the use of the system-supplied overlay facility, LOCAL or Load-On-CALl, for subprograms in Phase IV and Phase V (6). (See Figure 7 for an explanation of the Phase structure.) This added a degree of difficulty to moving the translator to a new machine as it must be adapted to the new overlay system. However, overlay features are available on most machines, especially ones with limited primary storage capacity. With the continuing enhancement of the translator to handle features not originally supported--CASE clauses, slicing and trimming, and loop clauses, for example--size had become an increasing problem. Two means of solution were obvious:

Phase	Name	Function
I	ALG01	Job control card recognition, translator option analysis and translator initialization.
II	ALG02	Source program input and listing and conversion to internal form.
III	ALG03	Keyword recognition, label declaration recognition, and construction of the block nesting table for Phase IV.
IV	ALG04	Source code parsing and code generation.
V	ALG05	Intrepretive executor.

Figure 7. Phase Structure of the OSU ALGOL 68 Translator

One was to restructure the translator in a major way to split off some of the tasks performed in Phase IV and perform them in earlier phases; the other solution was to continue to draw heavily upon the system overlay feature and multiple use of existing code segments, often at the loss of clarity. The first was rejected for this project as it would involve work requirements in excess of those obtainable during the time allowed. The second has been used heavily with the emphasis upon the former part (overlays) and not the latter (multiple code usage) when any loss of clarity could result. No size problems were encountered in Phase I, Phase II, or Phase III. However, they were ever present in Phase VI and Phase V. At one point it became necessary to remove a segment of code from Phase IV and replace it with a subroutine call be-

cause the IBM 1130 FORTRAN compiler's table capacity had been exceeded. Two subprograms were combined into one to allow a large block of code to be placed in LOCAL overlay; another section of code was moved from in-line to subroutine to provide space for necessary in-line code additions. All large segments of code which were required for the implementation of procedures were placed in subroutines designed to be used as LOCAL overlays. This policy had the unexpected advantage of improved clarity. In such cases code which already existed in another portion of the translator was duplicated because of accessibility problems posed by the overlay structure. The result is a more tree-structured program where a lattice structure might have developed otherwise.

Similar situations occurred in Phase V, the interpretive executor, where three subprograms were required to be in LOCAL overlays to provide room for the first additions. All code written was in modular form with the intent of using LOCAl overlays. The task was easier in this phase since the structure of the interpreter is basically a tree structure that decodes a pseudo-machine instruction which is a quadruple.

The cost for this approach has been an extremely slow compilation and execution process. The IBM 1130 is not an extremely fast machine. The OSU configuration has only one disk drive. The highly convoluted structure of the translator along with the extensive use of disk

files for compiler table storage results in a great deal of time being used for disk seeks. However, since the Oklahoma State University 1130 is operated on a hands-on, educational training basis, the performance of the ALGOL 68 translator is tolerable even if less than ideal.

Implementation Methods

Since the translator is a multi-phase translator, it was decided to effect the changes necessary for implementation of procedures in a phase-by-phase manner. Phase I (ALG01) performs an options analysis and initialization only; therefore, it has not been altered. The symbol table management technique employed in Phase IV make special treatment of the nesting structure analysis of the source program necessary. Since this analysis is repeated in three phases of the translator, they all have been modified.

As a space-saving measure, the symbol table used in this translator consists of a linear unordered table with one entry for each unique identifier of the program. Since multiple occurences of a single identifier to indicate different values in different blocks is quite allowable, the symbol table manipulation routines must provide for altering the single entry for that identifier when it is declared and also restoring the old entry when the range of that declaration is exited. This restoration is performed automatically upon block exit as indicated

by a close symbol in the input source code. The formal parameters pack provides a special problem when using this method. This pack is enclosed in parenthesis; however, if restoration is allowed when the close symbol is encountered, then the parameter declarations are not available when the routine to which they apply is parsed. To solve this problem, an implied block is created around the entire routine denotation-formal parameters pack, yielding moid and routine text. Then the block implied by the open and close symbols of the formal parameters pack is ignored.

This nest structure must be created in Phase II and Phase III also. Phase II prints a source listing which includes a nesting indicator if requested. Phase III produces the block nesting table used in Phase IV for label analysis. Therefore, the mechanisms required to syntactically analyze procedure declarations on a very basic scale have been added to both phases. Almost identical schemes are used in the two programs.

Phase IV (ALG04) performs the parse of the source code in internal form and generates all code for a unitary clause. A proper program is a specific type of unitary clause called a closed clause which is, through several reductions of the grammar composed of unitary clauses. In effect, Phase IV is a recursive program. This recursion is accomplished by stacking necessary translator values and reinitializaing these values as if a new parse were beginning. At the appropriate time, the stacked values

are retrieved. It is in this program that the first major enhancements have been made in order to support procedures.

The procedure declaration is a rather specific external object for which three subroutines, as well as in-line code to determine when to invoke these routines have been added to Phase IV to accomplish the new tasks imposed upon the translator. Figure 8 lists the subprograms and their purposes. The code for the procedure text is generated in sequence and is preceded by an unconditional branch generated by ALGPE. When the code is complete, ALGPT adds the address to the unconditional branch instruction. The code for the text of the routine is generated by Phase IV through the recursive process described above. Therefore, all of the facilities of the translator are available to the user when he writes a procedure.

The mechanism to parse the procedure call and the actual parameters pack which may accompany it are incorporated in one subroutine and in-line code in Phase IV. Since procedure variables are not supported, invocation can be assumed with the appearance of an identifier of mode proc at any location within the code except in an actual parameter list. Here either the routine or the value from its elaboration could be desired. The absence of a compile time descriptor has forced some arbitrary restrictions at this point. They have been indicated in the previous chapter. However, the requirement that any invocation of a procedure from within a parameter list be

Routine	Purpose
ALGPA	Performs analysis of the actual declarer for a proc, updates the symbol table to reflect the new declaration, and causes generation of the code for allocation of a descriptor at run-time.
ALGPE	Generates the implied block increase updates the symbol table for any parameters, generates the code for parameter retrieval, and parses and records the yielding moid.
ALGPT	Checks the result of the routine text against the yielding moid and generates code to coerce if necessary, generates return linkage, and removes the implied block along with restoring the symbol table.

Figure 8. Subprograms to Parse Procedure Declarations and Their Usage

"shielded" by a closed clause (enclosed in parentheses) does not seem a particulary harsh one.

The previously existing mechanisms for processing collateral clauses allowed their appearance only as subscripts or as parameters for transput. It has been necessary to add the recognition of collateral clauses in actual parameters packs to these mechanisms. The translator has the facility to create an internal identifier for a temporary value if needed. This facility has been used in keeping track of actual parameter values until all parameters in a pack are elaborated. The facility is needed only for temporaries as other values already have an identifier.

When an actual parameter pack is encountered, the parameters are parsed in order. Any code needed to compute the parameter value is generated. If the parameter is a temporary, the subprogram ALGPU is invoked to generate the code to give that value an internal identifier name. As the pack is being parsed, the identifier, internal or external, for each parameter is left on the translators parsing stack. When the last parameter is parsed, the Load Return Information instruction is generated. Then the identifiers are popped from the parsing stack, one-by-one, and a load parameter instruction is generated for each one. The process is completed by the generation of an Invoke Procedure instruction and insertion of the return address into the already generated Load Return Information instruction.

Eight new instructions may be generated during the analysis of a procedure declaration and the invocation of that procedure. These are summarized in Figure 9. Also, one existing instruction has been modified to support the allocation of storage for a procedure runtime descriptor.

The modifications to Phase V (ALGO5), the pseudo-machine, consist of a slight alteration of the existing routine to allocate storage, the addition of in-line code to decipher the added op codes, and subroutines to perform the functions required by the added instructions (see Figure 9).

The runtime descriptor maintained for a procedure is shown in Figure 10. It consists of five one-word entires.

```
RETRIEVE PARAMETER
RTP       800,R2,R3,...     R2 is the identifier number
                            R3 is the MODE including REF code
                            If R2 is zero retrieve Flag(s)

COMPLETE PROC DESCRIPTOR
CD        801,R2,R3,R4      R2 is the identifier number
                            R3 is 1 for completing static
                            information, is 2 for completing
                            entry point
                            R4 is the entry point if appro-
                            priate

PROC ENTRY
PRCIN     810,...,...,...

PROC EXIT
PEXIT     820,R2,R3,...     R2 is MODE of returned value
                            R3 is the number of rows

LOAD RETURN INFORMATION
LDINF     830,...,...,...

INVOKE PROCEDURE
INVOK     835,R2,...,...    R2 is the identifier number

SAVE SYMBOL TABLE
SYST      840,...,...,...

ALLOCATE PROC DESCRIPTOR
ALPD      040,R2,...,R4     R2 is 8
                            R4 is the identifier number
```

Figure 9. Additions to the Pseudo-Machine Instruction Set

Block Number	Global Display	Symbol Table Copy Key	Not Used	Entry Point

Figure 10. Procedure Runtime Descriptor

The block number is the block nest level which is one greater than the level in which declared. The global display is the address of the display which is active when the declaration is elaborated. The symbol table copy key is the key to the record of a disk file into which the runtime symbol table for the containing block is stored after all declarations are elaborated. These three entries are necessary to establish the global environment for the procedure when it is invoked. The last entry is the address of the first executable instruction of the routine. The storage for this descriptor is allocated through the use of the ALPD op code instruction which uses the routine ALGAS. The first three entries are completed when the identifier is declared to possess a value of mode proc. The last is completed when the routine denotation is encountered. This is compatible with the processes required if procedure variables were handled, and allows for future enhancement. Both of these functions are handled by the subprogram ALGPD (see Figure 11). The unused entry in the descriptor is supplied to allow for implementation of row-of proc at a later date.

Subroutine	
ALGPB	Performs the tasks required at procedure entry including establishing the new environment
ALGPC	Performs the load return information instruction
ALGPD	Performs the complete procedure descriptor instruction
ALGPG	Performs the save symbol table instruction
ALGPJ	Performs the invoke procedure instruction
ALGPP	Performs the retrieve parameter instructions
ALGBE	Performs the tasks required at procedure exit including return of the yield.
ALGPF	Performs the load parameter instruction

Figure 11. Subroutines Added to Phase V and Their Functions

The construction of the proper environment for execution of a procedure requires access both to the display which was active when the procedure was declared and to the runtime symbol table for that same block. The display creates no real problem as it exists in its needed form at any time when the procedure can be invoked.

The symbol table, however is dynamically altered with every block entry and block exit. To allow fast reconstruction of the necessary environment, the symbol

table is stored on disk when a block which contains a procedure declaration is executed. The location of this copy is recorded in the descriptor for the procedure. Only one copy is made for each block elaboration even though more than one procedure is declared. This is accomplished by execution of the instruction to save the symbol table, SVST, which is performed by the subprogram ALGPG. An additional mechanism was added to Phase V to manage the symbol table storage file in a dynamic manner.

The procedure entry instruction, PRCIN, is executed when a procedure routine is elaborated to establish the proper environment for the routine. It saves the old symbol table in the symbol table storage file and retrieves the one which is global to the procedure. It creates a new display using the global display as a pattern. The information necessary for this is included in the procedure descriptor which is placed on the stack prior to invocation. The subprogram which performs these tasks is ALGPB.

The subprogram ALGBE performs the analogous reverse tasks upon exit from a procedure. The old environment must be restored. Additionally any returned value must be placed on the top of the stack for the block to which control is returning. The instruction for procedure exit, PEXIT, appears as the last instruction of every procedure and initiates these actions. (This same program performs block exits).

A procedure may have zero or more parameters. These must be retrieved when elaboration of the routine begins. The retrieve parameter instruction, RTP, invokes the subprogram ALGPP in Phase V. This instruction has two forms. One indicates the retrieval of an actual value for a formal parameter. The other indicates retrieval of the parameter flag, a signal that parameter retrieval is complete. This is used as an error checking device since no error checks for parameter usage are made at compile time. The instruction of the first form contains the identifier number and mode information about the formal parameter. The parameter information on the runtime stack is a symbol table-like entry (address and mode). The mode information in the instruction indicates whether the parameter is ref amode or amode. Valves of mode amode are copied into local storage. Valves of mode ref amode are not copied; the existing value is used. Mode checks are made to insure that all passed parameters match expected parameters or that proper coercion can be performed.

The elaboration of a call must prepare all of the information about return, parameters, and invocation required by the procedure being called. This information is placed on top of the runtime stack during the elaboration of a call. The first instruction encountered in this process is the load return information instruction, LDINF. This causes the return address and the information required to restore the current environment (the display pointer

and block number) to be loaded on top of the stack with the parameter list flag on last. This flag signals the end of the parameter list to the routine code for parameter retrieval. Then, a load parameter instruction is executed for each parameter passed. (Any computation of the parameter value was done before the LDINF instruction). The symbol table entry for the actual parameter indentifier (external or internal) is copied onto the top of the stack. When all parameters are processed, an invoke procedure instruction, INVOK, will be executed which causes the runtime descriptor for the procedure indicated to be loaded and a branch to the address on top of the stack to occur. These functions are performed by the subprograms ALGPC, ALGPF, and ALGPJ, respectively. Figure 12 shows the stack top when elaboration of the procedure commences.

The elaboration of instructions which comprise the routine text is handled by facilities already existing in the translator except as they involve the use of the procedure facilities with which this effort is concerned.

All of the subprograms described above are written so that they can be included as overlays because of the space problems discussed earlier. This adds to the time problems of interpretive execution. However, the instructions involved do not comprise an extremely large portion of the object code for any reasonable program. Additionally, an effort has been made to perform as much work as

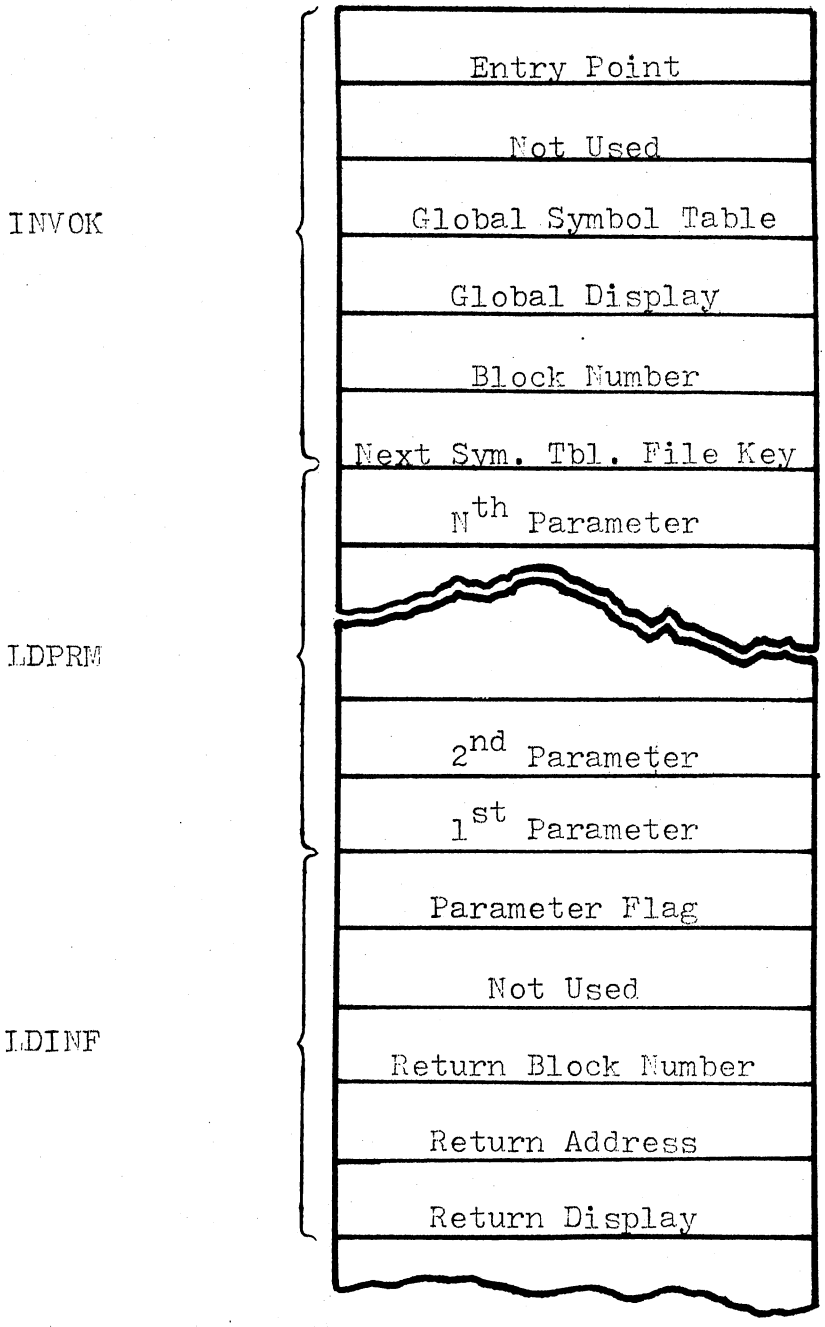

Figure 12. Top of the Runtime Storage Stack When Procedure Elaboration Begins and the Instructions Which Created It

reasonably possible in a single instruction, thereby reducing the overhead costs of instruction retrieval and decoding. To this end the subprogram to retrieve parameters, ALGPP, has the facility to retrieve successive instructions until the retrieve flag instruction is processed. This facilitates execution of these instructions which always occur in sets; however, there is some loss of continuity because of the break in the normal fetch/execute cycle.

CHAPTER IV

SUMMARY AND CONCLUSIONS

An omnipresent facet of almost every programming effort is the time/space trade-off. Rarely is a savings in space requirements accompanied by an increase in performance. In the current effort space was not just a consideration, it was an unalterable restriction. There was no more space in which to expand, therefore the program expanded in time. The methods outlined have been used successfully to implement the procedure features described. This provides the user with a new degree of power in his programming. New realms are opened for investigation. Perhaps most importantly, the programmer is provided the capacity which was not available previously for modularity in the design of his program.

The existing Oklahoma State University AIGOL 68 Translator has been enhanced to do the following:

1) syntactically recognize procedure constant declarations and calls to procedures;

2) generate the pseudo-machine code required to elaborate the declaration of procedure constants, calls to procedures, and the procedure routines themselves;

3) perform elaboration of the pseudo-machine instructions added as a result of enhancement.

These achievements provide the ALGOL 68 user at CSU with a very desirable facility. The cost is considerably slower compilation and execution of all jobs, whether or not the jobs use the new facility. However, the the added time requirement is a result of the space constraints imposed by the hardware system used and is not an inherent feature of the enhancements. On a machine where primary storage is more plentiful, only a slight time cost would be incurred by these changes.

Future extensions of the work can accomplish further additions only at an extremely high cost in the time required for a job to compile and execute. New organization of task performance needs to be achieved before significant enhancement will be desirable. A possible consideration would be the recognition of declarations at an earlier stage. This could be done in Phase III or in a separate phase between Phase III and Phase IV. This would remove a large burden from the extremely overburdened Phase IV.

A project worth consideration is the optimization of block structuring. The runtime overhead of block entry is enormous in the interpretive pseudo-machine. This overhead is of value only if a block containing declarations is entered, but is performed for every parenthesis set encountered except parameters packs. This could be easily done if declarations were recognized at an earlier time.

The implementation of formatted transput is

an effort currently underway for a version of the same translator which runs under OS/360 MVT on the IBM 360/65 at Oklahoma State University. This could contribute greatly toward expanding the usability of ALGOL 68 at this installation.

SELECTED BIBLIOGRAPHY

(1) Backus, J. W. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." Proceedings of the International Conference on Information Processing, UNESCO, Paris, 1959. Munich: R. Oldenbourg, 1960, pp. 125-132.

(2) Currie, I. F., S. G. Bond, J. D. Morison. "ALGOL 68-R." ALGOL 68 Implementation. J. E. L. Peck(ed.). Amsterdam: North-Holland Publishing Co., 1971, pp. 21-34.

(3) Goos, G. "Some Problems in Compiling ALGOL 68." ALGOL 68 Implementation. J. E. L. Peck(ed.). Amsterdam: North-Holland Publishing Co., 1971, pp. 179-196.

(4) Gries, D. Compiler Construction for Digital Computers. New York: John Wiley & Sons, Inc., 1971.

(5) Hedrick, G. E. and B. R. Alexander. "A Transition from PL/1 to ALGOL 68." Proceedings of the Second Vancouver Conference on ALGOL 68. Vancouver: University of British Columbia, 1972.

(6) IBM 1130 Disk Monitor System, Version 2, Programmer's and Operators Guide (GC26-3717).

(7) Jensen, John C. "Implementation of a Scientific Subset of ALGOL 68." (Unpublished Masters thesis, Oklahoma State University, 1973.)

(8) Ledin, George Jr. "Partial Status List of ALGOL 68 Implementation Activities." Proceedings of the San Francisco Conference on ALGOL 68 Implementation. San Francisco: University of San Francisco, 1973.

(9) Lindsey, C. H. "Final Report on Improvements to ALGOL 68." ALGOL Bulletin, No. 36 (Nov., 1973). pp. 8.

(10) Lindsey, C. H. and S. G. van der Meulen. Informal Intorduction to ALGOL 68. Amsterdam: North Holland Publishing Co., 1973.

(11) Manis, V. S. "ALGOL 68-V Run Time Organization." Proceedings of the Second Vancouver Conference on ALGOL 68 Implementation. Vancouver: University of British Columbia, 1972.

(12) Mouldry, J., J. Kral, J. Madrchal, J. Sokol. "Recognition of Routine-Denotations in ALGOL 68." ALGOL Bulletin, No. 35 (Mar, 1973), pp.32.

(13) Naur, P. (Ed.) "Report on the Algorithmic Language ALGOL 60." Communications of the ACM, Vol. 3, No. 5 (May, 1960), pp. 299-314.

(14) Naur, P. (Ed.) "Revised Report on the Algorithmic Language ALGOL 60." Communications of the ACM, Vol. 6, No. 1 (Jan, 1963), pp. 1-17.

(15) Peck, J. E. L. (Ed.) ALGOL 68 Implementation. Amsterdam: North-Holland Publishing Company, 1971.

(16) Peck, J. E. L. An ALGOL 68 Companion. Vancouver: University of British Columbia, 1971.

(17) Proceedings of an Informal Conference on the Implementation of ALGOL 68. Vancouver: University of British Columbia, 1969.

(18) Proceedings of the Oklahoma State University Conference on the Implementation of ALGOL 68. Stillwater, OK: Oklahoma State University, to be published.

(19) Proceedings of the San Francisco Conference on ALGOL 68 Implementation. San Francisco: University of San Francisco, 1973.

(20) Proceedings of the Second Vancouver Conference on ALGOL 68 Implementation. Vancouver: Universtiy of British Columbia, 1972.

(21) Sammet, Jean E. Programming Languages: History and Fundamentals. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1969.

(22) Van Doren, J. R. "A Conceptual Model for Dynamic Storage Administration in Block Structured Languages." Technical Notes, Stillwater , OK: Oklahoma State University, 1972.

(23) van Wijngaarden, A. (Ed.), B. J. Mailloux, J. E. L. Peck and C. H. Koster. "Report on the Algorithmic Language ALGOL 68." Numerische Mathematik, Vol. 14 (1969), pp. 70-218.

(24) van Wijngaarden, A. (Ed.), B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. "Revised Report on the Algorithmic Language ALGOL 68." Supplement to ALGOL Bulletin No. 36. Vancouver: University of British Columbia, 1974.

APPENDIX A

DEFINITION OF ALGOL 68 TERMS

The following is a list of terms unique to ALGOL 68 or which have a very specific meaning in reference to ALGOL 68. The definitions given here have been taken from Lindsey and van der Meulen (10). These definitions are not direct quotes in all cases. When the definition in Lindsey and van der Meulen involves ideas not needed for the present discussion, they have been parapharased or quoted in part.

Assignation - A type of unit consisting of a name yielded by the left hand side, a := symbol, and a value yielded by the right hand side. Its elaboration consists of making the name refer to the value.

Call - To initiate the elaboration of a procedure.

Coercend - An external object which may be coerced.

Coersion - The changing of the mode of a coercend to that required by its context.

Constant - An instance of a value which is immediately possessed by an external object or a coercend which possess or yields a value which is not a name.

Declaration - A phrase which causes a identifier to possess a value (maybe undefined).

Declarer - An external object which specifies some mode.

Denotation - Those entities which, in earlier languages, would have been known as "literals" or "constants".

Descriptor - An internal object kept for each multiple value and subvalue to record the values of its bounds.

Destination - The left hand side of an assignation.

Dyadic operator - An operator between two operands.

Elaboration - The process of inspecting an external object and causing the corresponding actions (as specified by the semantics of the Report) to take place.

External objects - A part of the program text.

Identifier - An external object consisting of a sequence of letters and digits with a leading letter used to possess a value.

Internal object - An object which is stored and manipulated inside the computer during elaboration.

Mode - The property of a value which defines the class to which it belongs, i.e. the amount of storage space it requires, the ways it may be manipulated, etc.

Moid - mode or void.

Monadic operator - An operator applied to the following operand.

Multiple values - A value consisting of a sequence of values, its "elements" of some (same) mode, together with a descriptor.

Parameter - An external object which possesses a value to be supplied to a procedure (actual-parameter) or which specifies the mode of a value required by the procedure (formal-parameter).

Parametrize - To substitute actual-parameters.

Phrase - The constituents which when separated by goon-symbols form a serial clause; statements.

Possess - The verb used to indicate the relationship between an external object and an internal object.

Prefixes - Symbols used to construct declarers or to indicate all the modes of the appropriate class, or to indicate values of those classes of modes; ref, [ ] (row of), proc, or void.

Primitive modes - The built-in modes in terms of which all other modes may be constructed; int, real, bool, char, or format.

Primitive values - A value of mode int, real, bool, char, or format.

Procedure - A coercend (usually an identifier or a routine-denotation which yields a value of a proc mode.

Program - The program text provided by the user, together with the standard-prelude and standard-postlude.

Range - A piece of program text which demarcates the scope of the variables which are locally generated during its elaboration.

Reach - A range, with the exclusion of all ranges contained within it.

Refer - The verb used to indicate the relationship between a name and a value.

Repetitive statement - for xxxx from xxxx by xxxx to xxxx while xxxx do xxxxxxxx. In the Revised Report (24) this is termed a loop clause and has a slightly different syntax.

Routine - The internal equivalent of a clause which is of

a proc mode.

Routine-denotation - The external object used to create a routine.

Scope - The technical term for the range in which a value is available for use.

Slice - A external object which possesses a subvalue.

Source - The right hand side of an assignation.

Standard-postlude - The administration of the completion of the program.

Standard-prelude - The declarations already built into the language.

Subvalues - A subset of the elements of a multiple, as specified by a different descriptor.

Symbol - The smallest external object, out of which all the others are constructed, e.g. a, +, begin, etc.

Transput - Input or output.

Variable - An instance of a value to which a name refers (so that it can be changed), together with that name; also, a coercend which possesses or yields a name.

Yield - The production of a result through elaboration.

APPENDIX B

READING THE GRAMMARS

Each rule of the grammars in this paper consists of four items: the left hand side, the connector, the right hand side, and the terminator.

The left hand side consists of one non-terminal symbol of the grammar.

The connector is a colon (:).

The right hand side consists of one or more alternative productions for the left hand side non-terminal. A semicolon (;) is used to separate alternatives and may be read as "or". Each alternative may consist of one or more symbols of the grammar, either ternimal or non-terminal. A symbol of the grammar may contain blanks. Therefore, if more than one symbol appears in an alternative, they are separated by a comma (,) which may be read as "followed by".

The terminator is a period (.).

All terminal symbols are distinguished by the appearance of the character sequence "symbol" as the last portion of the symbol.

The example given in Figure 13 illustrates the construction of grammar rules in this manner.

```
identifier:  letter, idtail.

idtail:  letter, idtail; digit, idtail; break symbol,
         idtail; empty.

letter:  asymbol; bsymbol; csymbol; dsymbol; esymbol;
         fsymbol; gsymbol; hsymbol; isymbol; jsymbol;
         ksymbol; lsymbol; msymbol; nsymbol; osymbol;
         psymbol; qsymbol; rsymbol; ssymbol; tsymbol;
         usymbol; vsymbol; wsymbol; xsymbol; ysymbol;
         zsymbol;.

digit:   0symbol; 1symbol; 2symbol; 3symbol; 4symbol;
         5symbol; 6symbol; 7symbol; 8symbol; 9symbol.
```

Figure 13. Grammar For an Identifier

APPENDIX C

REPRESENTATIONS FOR TERMINAL SYMBOLS IN THE GRAMMARS

This appendix contains the EBCDIC code symbols which are accepted for the terminal symbols of the grammars used in this paper.

procedure symbol	PROC
is defined as symbol	=
go on symbol	; or .,
routine symbol	: or ..
open symbol	(or BEGIN
close symbol	) or END
sub symbol	(/
bus symbol	/)
ref symbol	REF
comma symbol	,
integral symbol	INT
real symbol	REAL
complex symbol	COMPL
boolean symbol	BOOL
character symbol	CHAR
void symbol	VOID
break symbol	-

APPENDIX D

ALGOL 68 USERS' GUIDE TO PROCEDURES

Introduction

The procedure facility available to the ALGOL 68 programmer at Oklahoma State University is a subset of the feature described in the Revised Report. Never the less, the current implmentation supplied a capability very much in line with that available in FORTRAN, PLAGO (a Polytechnic Institute of Brooklyn PL/I dialect translator), and other languages used for instructional purposes at CSU.

Presently the supported features are the declaration of procedure constants and the elaboration of these procedure constants by a call. The more powerful features such as procedure variable ref proc, row-of proc, and deproceduring are not supported. Each of the two supported features is discussed in greater detail in one of the following sections.

Procedure Declarations

The Oklahoma State University ALGOL 68 Translator supports the declaration of routine constants. The declaration of a routine constant causes an identifier to possess an internal object of the mode proc in the terms of the Revised Report. From the users' point of view, it is a means of creating a series of statements

which will perform a specific task and the ability to cause those statements to be executed from one or more remote positions in the program. Additionally, some of the data needed by the series of statements can be supplied as parameters from the remote location. This provides the user with efficiency because often-used code need not be repeated and flexibility because different data can be used at different times.

The general form for constructing a procedure declaration is given in Figure 14. The identifier is any non-keyword character string beginning with a letter and consisting of letters digits or the break character (_). It must be unique in its first 8 characters. The parameter declarations and the enclosing parentheses are optional. If not needed, they may be omitted. If present, any number of parameters may be declared--each separated from successive ones by commas. Each parameter declaration consists of a virtual mode and identifier. The allowable modes are INT, REAL, COMPL, BOOL, CHAR, or PROC. Multiples, of all but PROC are allowed; however, actual bounds may not be specified. If actual bounds are present,

```
PROC identifier=(parameter declarations), return mode:
                routine text;
```

Figure 14. General Form of a Procedure Constant Declaration

a warning message is printed and the values are ignored. Also, one level of REF may be applied to any mode or row-of mode except PROC. For example, the parameter declarations INT J, [ ] CHAR NAME, and REF [,] BOOL CMATRIX are all acceptable; however [0:12] REAL VECTOR, REF PROC REAL PVARIAB and [ ] PROC REAL SWITCH are not. (The user is referred to the section "Procedures as Parameters" of this appendix for a full discussion of that subject.)

The effect of one level of REF is to allow any changes which occur in the value of the parameter during elaboration to occur also in the actual parameter value. This is referred to as "side effect" or call-by-reference.

The return mode must be present in all cases. If no return mode is needed, the keyword VOID must be specified. Otherwise, INT, REAL, COMPL, BOOL, and CHAR as well as the row-of mode formed from these is valid.

The routine text is a unitary clause. It need not be enclosed in parentheses unless it consists of more than one statement (contains a semicolon). (The case of a routine consisting only of a declaration could serve no purpose and would not be valid.)

Some degree of abbreviation is allowed in the declaration of more than one parameter of the same mode. In this case, the mode need not be repeated for each parameter. The mode is specified for the first parameter and omitted for consecutive parameters for which the same mode is desired. This results in a construction such as AMODE

ID1, ID2, ID3 where three parameters of mode AMODE are declared. Figure 15 contains examples of several valid procedure constant declarations.

```
PROC A = (INT J, REF REAL R)REAL: R:=R**J;
PROC B = VOID: PRINT(NULLCHAR);
PROC C = (REF(/ /)CHAR STRG, INT CODE)INT: (...);
PROC D = (INT J,K,L,REF COMPL X,Y)VOID:  (...);
PROC E = (INT M, REAL VAL)(/ /)INT: (...);
```

Figure 15. Valid Procedure Constant Declarations

Procedure Calls

A procedure call is the mechanism by which the routine created with a procedure declaration is used. Where a procedure call may appear in a program is controlled by a very complex set of rules. However, with the present implementation the user is fairly free to use a procedure call anywhere he could use a constant of the mode yielded by the routine. There are two exceptions to this: one is in an identity, or initialized declaration, of a value of a mode other than proc; the other is in the actual parameters pack of a procedure call. However, this latter restriction can be circumvented by enclosing the call in parentheses in the paramenters pack

(see Figure 17).

Figure 16 shows the general form of a procedure call. The identifier must be a valid identifier (as defined in Appendix A) for which a procedure declaration has already appeared. The scope of that declaration must include the call. The structure of the actual parameters list is dependent upon the formal parameters list in the declaration. If no parameters pack appears in the declaration, then none should appear in the call. If there are paramenters required, the actual parameter list must supply an actual value for each formal parameter. The actual value must match the formal parameter in mode or be coercible to that mode. Since the parameter declarations in the formal parameters pack is a formal declaration, the REF which may appear there is implied by all actual declarations. Therefore, the actual value declared INT is acceptable to the formal parameter declared REF INT. However, the actual value declared INT is not acceptable to a formal parameter declared REF REAL. Only a value declared REAL would be acceptable to that formal parameter.

The result of declaring a parameter to be REF AMODE rather than AMODE (where AMODE represents any of the five standard modes or their multiples) is to allow "side effect" from the procedure. This implementation allows the user to change the value referred to by a formal identifier of mode AMODE. However, he is assured that the value of the actual parameter is not changed (call

```
identifier (parameters list)
```

Figure 16. General Form For a Procedure Call

by-value is used). On the other hand for a formal parameter of mode REF AMODE the user is assured that any change in the value of the parameter will cause a change in the value of the actual parameter. (Note. The user should avoid changing the value of a formal parameter of mode AMODE. This is not allowed in the strict language of the Report and probably not supported by most compilers.)

No compile time parameter correspondence checks are made. The user should be quite careful in constructing calls as any errors will not be detected until runtime. Figure 17 contains examples of several valid calls.

```
PRINT (("THE ANSWER IS", S:=12.0*A(J,B)));
B;
I:=3 + C(NAME,12);
D(JJ,KK,LL,CX CY);
E(12,(A(KK,B)))
```

Figure 17. Valid Calls For a Procedures of Figure 15.

Procedures as Parameters

The current implementation allows the user to declare a procedure to be a formal parameter and to pass a procedure name as an actual parameter. To declare a parameter of mode PROC requires a formal declaration as illustrated in Figure 18. The virtual parameter list and enclosing parentheses are omitted if the procedure parameter is to have no parameters itself. A virtual parameter list is identical to a formal parameter list described earlier except no identifiers are present. The return mode is specified exactly as it is for an actual procedure constant declaration. The identifier is the formal parameter which represents the actual procedure parameter.

When calling a procedure which has a procedure as one of its parameters, the user should specify the identifier for a previously declared procedure constant (whose scope includes the call) without any actual parameters pack and <u>not</u> enclosed in parentheses in the actual parameters pack of the call. Figure 19 contains an example.

PROC (virtual parameter list) return mode identifier

Figure 18. General Form For a Formal Procedure Parameter Declaration

```
BEGIN

      REAL VAL; INT J;
      PROC B = (INT N, PROC(REAL)REAL CUBE)REAL:CUBE(N);
      PROC A = (REAL R)REAL:R**3;
      READ (J);
      VAL:= V(J,A);
      PRINT (VAL)

END
```

Figure 19. Passing the Procedure A as an Actual Parameter to B

APPENDIX E

SAMPLE PROGRAMS

```
// JOB
// XEQ ALGOL
:JOB **********
```

```
STMNT   NEST   SOURCE          PR   ALGOL 68   PR

   1           BEGIN
   1      1            COMMENT
   1      1            THIS PROGRAM ILLUSTRATES THE
   1      1            DECLARATION OF A SIMPLE PROCEDURE
   1      1            AND ITS INVOCATION.
   1      1            COMMENT
   1      1       PROC A = (INT J) REAL:
   1      2         BEGIN
   1      3            PROC B = INT: J;
   2      3            REAL I;
   3      3            I:=J;
   4      3            PRINT (("I = ",I));
   5      3            I
   5      3       END;
   6      1       REAL KK;
   7      1       KK:=A(3);
   8      1       PRINT (KK);
   9      1       A(1)
   9      1    END
```

```
:ENTRY
I =
 0.30000E 01
 0.30000E 01

I =
 0.10000E 01
```

```
// JOB
// XEQ ALGOL
:JOB ******************

STMNT   NEST   SOURCE          PR   ALGOL 68   PR

   1              BEGIN
   1       1           COMMENT
   1       1           THIS PROGRAM TESTS THE DIRECT RECURSIVE
   1       1           INVOCATION OF A PROCEDURE.
   1       1           COMMENT
   1       1           INT N;
   2       1           PROC FACTORIAL = (INT N) REAL:
   2       2                BEGIN
   2       3                     REAL PROD;
   3       3                     PROD := (N=1 | 1.0 | N*FACT-
   3       3                     TORIAL(N-1) );
   4       3                     PRINT(( N, "FACTORIAL IS",
   4       3                     PROD, NEWLINE));
   5       3                     PROD
   5       3                END;
   6       1           READ(N);
   7       1           IF N>0
   7       2           THEN PRINT (( N, "FACTORIAL IS",
   7       2           FACTORIAL(N) ))
   7       2           ELSE PRINT (( N, "FACTORIAL IS
   7       2           UNDEFINED"))
   7       2           FI
   7       1      END

:ENTRY
      6
FACTORIAL IS
      1
FACTORIAL IS 0.10000E 01
      2
FACTORIAL IS 0.20000E 01
      3
FACTORIAL IS 0.60000E 01
      4
FACTORIAL IS 0.24000E 02
      5
FACTORIAL IS 0.12000E 03
      6
FACTORIAL IS 0.72000E 03

             0.72000E 03
```

```
// JOB

// XEQ ALGOL

:JOB ****************

STMNT   NEST   SOURCE          PR   ALGOL 68   PR

   1              BEGIN
   1       1           COMMENT
   1       1           THIS PROGRAM ILLUSTRATES A BRANCH TO A
   1       1           LABEL OUTSIDE OF THE PROCEDURE.
   1       1           COMMENT
   1       1           PROC A = VOID: QUIT;
   2       1           PRINT("ONLY ONE MESSAGE SHOULD FOLLOW
   2       1           THIS MESSAGE:);
   3       1           A;
   4       1           PRINT (" THIS MESSAGE SHOULD NOT BE
   5       1           PRINTED ");
   5       1           PRINT(" THIS MESSAGE SHOULD BE
   5       1   QUIT:   PRINTED!!!")
                   END

:ENTRY
ONLY ONE MESSAGE SHOULD FOLLOW THIS MESSAGE
 THIS MESSAGE SHOULD BE PRINTED!!!
```

```
// JOB
// XEQ ALGOL
:JOB ****************
STMNT   NEST   SOURCE          PR   ALGOL 68   PR

   1              BEGIN
   1       1           COMMENT
   1       1           THIS PROGRAM TESTS THE FOLLOWING
   1       1           THREE FUNCTIONS:
   1       1               1)  PASSING A PROC AS A PARAMETER
   1       1               2)  INVOKING A PROC PARAMETER
   1       1               3)  INDIRECT RECURSION.
   1       1           COMMENT
   1       1           PROC D = (PROC VOID Q) VOID:
   1       2                BEGIN
   1       3                     PRINT("START D");
   2       3                     Q;
   3       3                     PRINT("END D")
   3       3                END;
   4       1           PROC A = VOID:
   4       2                BEGIN
   4       3                     PROC B = VOID: PRINT("B");
   5       3                     PROC C = VOID: PRINT("C");
   6       3                     PRINT("START A");
   7       3                     D(C);
   8       3                     B;
   9       3                     PRINT("END A")
   9       3                END;
  10       1           A;
  11       1           D(A)
  11       1      END

:ENTRY
START A
START D
C
END D
B
END A
START D
START A
START D
C
END D
B
END A
END D
```

```
// JOB
// XEQ ALGOL
:JOB ****************

STMNT   NEST   SOURCE              PR   ALGOL  68   PR

   1           BEGIN
   1      1              COMMENT
   1      1              THIS PROCEDURE TESTS THE MANAGEMENT
   1      1              OF THE ENVIRON FOR THE PROCEDURE.
   1      1              THE CORRECT SOLUTION IF 14 FOR ALL
   1      1              ELEMENTS OF THE ARRAY.
   1      1              COMMENT
   1      1         PROC P=(PROC VOID X, INT Y) VOID:
   1      2          BEGIN
   1      3           PROC Q=(PROC(REF INT) VOID Z)VOID:
   1      4            BEGIN
   1      5             (/1:10/) INT F:
   3      5             F(/1/):=13;
   3      5             Z(F(/1/)+Y)
   3      5            END;
   4      3           Q(X)
   4      3          END;
   5      1         PROC R=VOID:
   5      2          BEGIN
   5      3            INT I,
   6      3            (/1:10/) INT G;
   7      3            PROC U=(REF INT W) VOID: G(/I/):=W;
   7      3            #                                  #
   7      3            FOR J TO 10 DO
   7      3             BEGIN
   7      4              I:=J;
   8      4              G(/I/):=23;
   9      4              P(U,1)
   9      4             END;
  10      3            PRINT(NEWLINE,NEWLINE,NEWLINE,"GLOBAL
  10      3            DISPLAY TEST");
  11      3            PRINT(G)
  11      3          END;
  12      1         R;
  13      1         PRINT("END GOLBAL DISPLAY TEST")
  13      1       END

:ENTRY

GLOBAL DISPLAY TEST
     14              14          14        14        14
     14              14          14        14        14
END GLOBAL DISPLAY TEST
```

```
// JOB
// XEQ ALGOL
:JOB ****************

STMNT   NEST   SOURCE          PR   ALGOL  68   PR

   1              BEGIN
   1       1            COMMENT
   1       1            THIS PROGRAM ILLUSTRATES THE PASSING
   1       1            OF A ROW-OF AMODE AS A PARAMETER.
   1       1            COMMENT
   1       1            PROC ARRAYINCRE = (REF (/ /) REAL A,
   1       1            REAL INC) VOID:
   1       2                 BEGIN
   1       3                      INT LB, UB;
   2       3                      LB := 1 LWB A;
   3       3                      UB := 1 UPB A;
   4       3                      FOR SUBSC FROM LB BY 1 TO
   4       3                      UB
   4       3                      DO
   4       3                           A(/SUBSC/) :=
   4       3                           A(/SUBSC/) + INC
   4       3                 END;
   5       1            INT LB, UB;    REAL INC;
   7       1            READ (( LB, UB, INC ));
   8       1            BEGIN
   8       2                 (/ LB:UB/) REAL ARRAY;
   9       2                 PRINT ( " THE ORIGINAL ARRAY");
  10       2                 FOR SUBSC FROM LB TO UB
  10       2                 DO
  10       2                      (
  10       3                           READ( ARRAY(/SUBSC/) );
  11       3                           PRINT ( ARRAY
  11       3                           (/ SUBSC /) )
  11       3                      );
  12       2                 ARRAYINCRE(ARRAY, INC);
  13       2                 PRINT ((NEWLINE, NEWLINE, "AFTER
  13       2                 INCREMENT BY ", INC ));
  14       2                 FOR SUBSC FROM LB TO UB
  14       2                 DO PRINT ( ARRAY(/ SUBSC /) );
  15       2            END
  15       1       END
```

```
:ENTRY
 THE ORIGINAL ARRAY
 0.25000E 01
 0.37500E 01
 0.71600E 01
-0.34000E 01
 0.00000E 00
 0.10050E 03

AFTER INCREMENT BY
 0.50000E 01
 0.75000E 01
 0.87500E 01
 0.12160E 02
 0.16000E 01
 0.50000E 01
 0.10550E 03
```

```
// JOB
// XEQ ALGOL
:JOB **********
```

```
STMNT   NEST   SOURCE          PR   ALGOL 68  PR

   1           BEGIN
   1      1            COMMENT
   1      1            THIS PROGRAM PERFORMS THE SYNTACTIC
   1      1            ANALYSIS OF SIMPLE EXPRESSIONS.
   1      1            THE INPUT EXPRESSIONS MUST BE ONE PER
   1      1            CARD STARTING IN COLUMN ONE.
   1      1            THE PROGRAM IS TERMINATED BY THE
   1      1            APPEARANCE OF A SEMICOLON IN COLUMN
   1      1            ONE OF THE INPUT CARD.
   1      1            COMMENT
   1      1        INT I:=1;
   2      1        (/20/) CHAR CARD;
   3      1    PROC LETTER = BOOL:
   3      2        BEGIN
   3      3        IF CARD(/I/) = "X" OR
   3      4            CARD(/I/) = "Y" OR
   3      4            CARD(/I/) ="Z" THEN I +:=1;
   4      4                                TRUE
   4      4                            ELSE FALSE FI END;
   5      1        PROC DIGIT = BOOL:
   5      2            BEGIN
   5      3            IF CARD(/I/) > = "0" AND
   5      4                CARD(/I/) < = "9" THEN
   5      4                    I +:= 1;
   6      4                    TRUE
   6      4                ELSE FALSE FI END;
   7      1        PROC NUMBER = BOOL:
   7      2            BEGIN
   7      3            IF DIGIT THEN IF NUMBER THEN TRUE ELSE
   7      3            TRUE FI
   7      4                ELSE FALSE FI END;
   8      1        PROC NAMETAIL = BOOL:
   8      2            BEGIN
   8      3            IF DIGIT
   8      4                THEN IF NAMETAIL THEN TRUE ELSE
   8      4                TRUE FI ELSE IF LETTER
   8      5                    THEN IF NAMETALL THEN TRUE
   8      5                    ELSE TRUE FI ELSE FALSE FI
   8      5                    FI END;
   9      1        PROC NAME = BOOL:
   9      2            BEGIN
   9      3            IF LETTER THEN IF NAMETAIL THEN TRUE
   9      3            ELSE TRUE FI ELSE FALSE FI END:
```

```
STMNT   NEST   SOURCE

  10      1        PROC P = BOOL:
  10      2            BEGIN
  10      3            IF NAME THEN TRUE ELSE IF NUMBER THEN
  10      3            TRUE ELSE FALSE FI FI END;
  11      1        PROC F = (PROC BOOL E) BOOL:
  11      2            BEGIN
  11      3            IF CARD(/I/) = "("
  11      4                THEN I +:= 1;
  12      4                    IF E THEN
  12      5                        IF CARD(/I/) = ")"
  12      6                            THEN I +:= 1;
  13      6                                TRUE
  13      6                            ELSE FALSE
  13      6                        FI
  13      5                    ELSE FALSE
  13      5                FI
  13      4                ELSE IF P THEN TRUE ELSE FALSE FI
  13      4            FI END;
  14      1        PROC T = (PROC BOOL E) BOOL:
  14      2            BEGIN
  14      3            IF F(E) THEN
  14      4                IF CARD(/I/) = "*"
  14      5                THEN
  14      5                    I +:= 1;
  15      5                  IF T(E) THEN TRUE
  15      6                      ELSE FALSE
  15      6                  FI
  15      5                ELSE TRUE
  15      5            FI
  15      4            ELSE FALSE
  15      4        FI END;
  16      1        PROC E = BOOL:
  16      2            BEGIN
  16      3            IF T(E) THEN
  16      4                IF CARD(/I/) = "+"
  16      5                    THEN
  16      5                        I +:= 1;
  17      5                        IF E THEN TRUE
  17      6                            ELSE FALSE
  17      6                        FI
  17      5                    ELSE TRUE
  17      5                FI
  17      4                ELSE FALSE
  17      4            FI END;
  18      1            READ (CARD);
  19      1            WHILE CARD(/1/) = ";" DO
  19      1            BEGIN
  19      2                PRINT (CARD);
```

```
STMNT   NEST   SOURCE

  20      2            IF E THEN IF CARD(/I/) = ";" THEN
  20      2            PRINT ("BALID")
  20      4         ELSE PRINT ("INVALID") FI ELSE PRINT
  20      4         ("INVALID") FI;
  21      2              READ (CARD);
  22      2            I := 1
  22      2            END
  22      1    END

:ENTRY
X1+X2*X3;
VALID
X1+(X2+X3*X4);
VALID
(((X)));
VALID
(((X));
INVALID
```

```
// JOB
// XEQ ALGOL
:JOB **********
```

```
STMNT   NEST   SOURCE          PR   ALGOL  68   PR

    1          BEGIN
    1      1           COMMENT
    1      1           THIS PROGRAM CONTAINS TWO PROCEDURES.
    1      1
    1      1           THE PROCEDURE 'BUILD' IS USED TO
    1      1           CONSTRUCT A BINARY TREE FROM ELEMENTS
    1      1           IN THE INPUT STREAM.
    1      1
    1      1           THE PROCEDURE 'TRAVERSE' IS USED TO
    1      1           PERFORM AN IN-ORDER TRAVERSAL OF THE
    1      1           TREE AND PRINT ITS CONTENTS.
    1      1           COMMENT
    1      1       INT NUMELEM;
    2      1       READ(NUMELEM);
    3      1       BEGIN
    3      2        (/NUMELEM/) INT KEY;
    4      2        (/NUMELEM,3/) INT TREE;
    5      2        INT I, DIR;
    6      2        INT START := 0;
    7      2        PROC BUILD = ( REF (/ /) INT KEY,
    7      3                       REF (/ , /) INT TREE) VOID:
    7      3            BEGIN
    7      4            INT NRKEY;
    8      4            NRKEY := UPB KEY;
    9      4            PRINT (("NRKEY = ",NRKEY));
   10      4            PRINT(NEWLINE);
   11      4            FOR J TO NRKEY DO
   11      4            BEGIN
   11      5                IF START = 0 THEN
   11      6                     START := J;
   12      6                    TREE(/J,1/) := KEY (/J/);
   13      6                    TREE(/J,2/) := 0;
   14      6                    TREE(/J,3/) := 0
   14      6                ELSE
   14      6                    BEGIN
   14      7                    INT K := START;
   15      7                    TREE(/J,1/) := KEY (/J/);
   16      7                    TREE(/J,2/) := 0;
   17      7                    TREE(/J,3/) := 0:
   18      7                    K := START;
```

```
STMNT   NEST   SOURCE

  19      7                    WHILE K >  0 DO
  19      7                    BEGIN
  19      8                            I := K;
  20      8                            IF KEY (/J/) <  TREE
  20      8                            (/K,1/)
  20      9                              THEN
  20      9                                 DIR := 0;
  21      9                                 K := TREE(/K,2/)
  21      9                            ELSE
  21      9                                 DIR := 1;
  22      9                                 K := TREE (/K,3/)
  22      9                            FI;
  23      8                    END;
  24      7                            IF DIR = 0 THEN
  24      8                                TREE(/I,2/) :=J
  24      8                            ELSE TREE(/I,3/) :=J
  24      8                            FI
  24      7                    END
  24      6                FI
  24      5            END
  24      4            END;
  25      2        PROC TRAVERSE = (INT START) VOID:
  25      3        BEGIN
  25      4            IF TREE (/START,2/) > 0 THEN TRAVERSE
  25      4            (TREE(/START,2/)) FI;
  26      4            PRINT (TREE(/START,1/));
  27      4            IF TREE(/START,3/) > 0 THEN TRAVERSE
  27      4            (TREE(/START,3/)) FI
  27      4        END;
  28      2        (/4/) INT T2;
  29      2        READ (KEY);
  30      2        BUILD(KEY,TREE);
  31      2        FOR S TO UPB KEY DO BEGIN
  31      3        T2(/1/) :=S;
  32      3        T2(/2:4/) := TREE(/S/);
  33      3        PRINT (T2)
  33      3        END;
  34      2        TRAVERSE (1)
  34      2        END
  34      1    END
```

```
:ENTRY
NRKEY =
     10

      1          35          2          3
      2          18          4          0
      3          77          0          6
      4          -3          0          5
      5          15          7          9
      6         363         10          0
      7           0          0          8
      8           1          0          0
      9          15          0          0
     10          85          0          0
     -3
      0
      1
     15
     15
     18
     35
     77
     85
    363
```

VITA

Alan Daniel Eyler

Candidate for the Degree of

Master of Science

Thesis: THE IMPLEMENTATION OF A SUBSET OF PROCEDURES IN AN ALGOL 68 COMPILER

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Pawnee, Oklahoma, July 24, 1947, the son of Mr. and Mrs. Ray B. Eyler.

Education: Graduated from Ralston High School, Ralston, Oklahoma, in May, 1965; received the degree of Bachelor of Science from Oklahoma State University, Stillwater, Oklahoma, in May; 1970, with a major in Physical Sciences; certified for teaching at the secondary level in Oklahoma; completed requirements for the Master of Science at Oklahoma State University in July, 1975.

Professional Experience: Graduate Teaching Assistant in the Computing and Information Sciences Department at Oklahoma State University from August, 1973 to May, 1975; member of the Association for Computing Machinery; President of the OSU Student Chapter of the ACM from August, 1974 until May, 1975.